# Tuning-Up MLP

A REPORT

Alireza Mahmoudian[1]

[1]SID: 95112095

# Chapter 1

# Introduction

## 1.1 Network Topology

MLP, short for Multi Layer Perceptron, is a feed-forward multilayer neural network, widely used with supervised learning methods to solve regression and classification problems. MLPs' power is due to their ability to *represent* any differentiable function, making them *general function approximator*s. So they can act as a regression function in regression problems, or as a *discriminator* curve in classification.

As a feed-forward network, MLP consists of layers of neurons, each fully connected to the next layer. Data sample vectors enter the network via the first layer, or *input layer.* Each attribute of the samples is taken as input by an input neuron. Or alternatively for discrete-valued attributes with $p$ possible values, a vector of length $p$ is given to $p$ neurons of the input layer, with all elements set to *0*, except for one element that determines the value of the discrete-valued attribute, which is set to *1.* Input neurons then compute *identity* as their activation and pass it to the next layer. So their only job is to feed the input pattern to the network.

The regression/classification result is then taken from the output of neurons in the last layer, or *output layer.* In different applications, our interpretation of the result determines how many neurons are there in the output layer, and what their output values mean. In a regression problems, the output layer has a single neuron, as a regression function would map any input to a single real value. But as a classifier, the MLP must assign the input pattern to one (or generally $k$)[1] of $n$ classes. So the output must be a vector of size $n$ whose greatest value(s) then defines the chosen class.[2]

Between these terminal layers, there are more layers with different number of neurons, which are called *hidden layers.* There must be at least two hidden layers so "multilayer" in MLP's name would be meaningful; In contrast with SLP, *Single Layer Perceptron*, which only differs in structure from MLP in number of such layers. Neurons in these intermediate layers are called "hidden" because our only interface with the network is either via the input we feed to input neurons or the output we take from output neurons,

[1] Alternatively, output neurons could have a step function as their activation, so they would give a result of 1 for all values greater than their respective threshold or 0 otherwise; An approach for a multi-label classifier. Although when it comes to learning, we will see that the identity function is preferred here too.

[2] Another possible classifier which uses a single output neuron was discussed at the class. We concluded then that it could be helpful only when classes have a real numerical relation, so we can assign a (region of) real value to each class. Since we do not have such condition in our benchmarks in this report, so we will omit this approach.

so we don't know anything about the state of other neurons; They are *hidden* from the user.

The main role of hidden layers is to give the network the ability to classify linearly inseparable data, what SLPs are not able to do. From a geometrical point of view, we know that a set of linearly inseparable points could be linearly separable in a higher dimension. In this regard, we can see what hidden neurons do as extracting hidden attributes of the dataset, as if they are adding dimensions to the data points and their activation functions are hyperplanes that discriminate points in this higher dimension.

Theoretically, It can be shown that if we use linear activation function for hidden neurons, not matter how many layers we add to the network, its power will not be increased further than a single perceptron model. For example, assume a neuron $j$ which (usually) has a propagation function of the form $net_j = \mathbf{w}_j\mathbf{x} + b$ where $\mathbf{x}$, $\mathbf{w}_j$ and $b$ are vectors of neuron $j$'s input from the last layer, vector of their respective synaptic weights and a bias value, respectively, and an activation function of linear form $x_j(net_j) = p * net_j + q$. So we have $x_j(net_j) = p(\mathbf{w}_j\mathbf{x} + b) + q$. It's obvious that we can take $p\mathbf{w}_j$ as a new weight vector and $pb + q$ as a new bias value; As we can simplify any polynomial to it's canonical form. Such neuron won't take us any further than a single neuron would, so we can safely omit it. This can be generalized to any number of layers with linear activation.

In fact, if all we care about is representability, we would choose *Heaviside step function* as activation function. But for the sake of *learnability*, as we'll see, a differentiable function is preferred. So we use some differentiable functions which approximate step function with a desired slope; Like *sigmoid function* or *tanh*. They will also alleviate the computational cost of the learning, as we will see, by simplicity of their derivatives.

Despite it's feed-forward topology, an MLP does not have shortcuts. This is also due to complexities in the learning algorithm we use.

## 1.2 Learning Algorithm

A *supervised* approach for teaching neural networks is to adjust synaptic weight so that the network output is near to the what we expect from it. This is equivalent to minimizing a distance (or error) function of network output and desired values. These desired values can be represented to the network by a set of labeled[3] data samples, which is called *training set*. We use *squared error* here as minimization objective function. We also use *gradient descend* technique for minimization purpose. It utilizes the fact that for every point on a function, a local minimum can be found at the opposite direction of the gradient vector of the function at that point. So the idea is that we can start from a random point and traverse the function in the opposite direction of the gradient vector in an iterative manner to reach a local minimum. We also need our network output and thus the minimization objective function, to be a function of synaptic weights, so that minimization algorithm gives us the optimal values for weights. Then the error in the last layer could be propagated to the latter layers so that we can have a minimization objective for hidden neurons too. This is why it was preferred not to use short cuts in the MLP's topology.

What was briefed above, is the whole idea of a learning algorithm called *backpropagation of error*, or simply backpropagation. It is a generalization of the *delta rule* for MLP networks, made possible by using the chain rule to iteratively compute gradients for each

[3]Each sample is assigned to a class (for classification) or given a value (for regression) by user.

layer. Now we see why we chose a differentiable activation function for all layers. Delta rule, as its name suggests, computes a *correction* value for weights, in the direction and amount that gradient descent proposes.

So we want to compute the gradient of the error function for a sample pattern and an initial (random) weight, then modify each synaptic weight by it, i.e. $\Delta w_{ij} = -\eta \frac{\partial E_j}{\partial w_{ij}}$, where $\frac{\partial E_j}{\partial w_{ij}}$ determines the direction of next iteration and $\eta$ is a factor to control the length of step we take proportional to the gradient vector length. As mentioned before, the error function we use for neuron $j$ is $E_j = \frac{1}{2}\sum_j (t_j - y_j)^2$. Since we don't have $w_{ij}$ as a direct variable of $E$, so we will apply derivation chain rule twice to reach a function of $w_{ij}$ which is, as seen before, $net_j$. So we will have $E_j = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_j}$. It can then simply calculated as $E = -(t_j - y_j(net_j))y'_j(net_j)x_i$. So we have the following rule for updating weight values of the last layer:

$$\Delta w_{ij} = \eta(t_j - y_j(net_j))y'_j(net_j)x_i$$

This rule then can be generalized, so weight adjustment for each neuron can be calculated from error in output of that neuron and inputs from the previous layer.

Backpropagation has also it's disadvantages. For example, since our step size in each iteration depends on the slope of the function at the current point (that is the length of the gradient vector), in vast plateaus our traverse could become real slow or conversely, when we are descending too fast, we would probably miss a local minimum by one single huge step. Another case can happen when the gradient becomes suddenly very close to zero, not at a good minimum of the function but at small plateaus in our way to descend. A method to prevent sudden variations int the amount of the step size is to keep a certain amount of the last adjustment value when calculating the next. We used this correcting term, which is known as *momentum* in our calculation of step size. So $\Delta w_{ij}$ for the step $k+1$ would become:

$$\Delta {w_{ij}}^{k+1} = \eta(t_j - y_j(net_j))y'_j(net_j)x_i + \alpha \Delta {w_{ij}}^{k+1}$$

where $\alpha$ is a constant factor. This would prevent $\Delta w_{ij}$ from changing quickly. Advantages of this method in comparison with the plain backpropagation is shown, and it is recommended by the textbook as well.

The iterative algorithm can keep on getting closer to a local minimum till meeting a stopping criterion. In this report the learning algorithm checks if the error function is below a certain $\epsilon$ and it will stop if it is so.

## 1.3 Benchmarks

For reasons we will see in the next sections, we performed experiments on two benchmarks. First we tried to tune up an MLP for classification of a dataset[4] of football match results. The dataset, consists of tuples for 106 matches in English premier league, has 8 numerical attribute, all of which has been normalized to the interval of [0, 1], demonstrating the quality of the home team in it's four lines of forward, middle, defense and the goal keeper, and the other four measure same parameters for the away team. The data is classified into three classes, and labeled by a vector of two 0s and a single 1 (of size three), where the position of the single one determines to which class the sample belongs. Three classes show if the game was a win for the home team, a tie or a win for the away team.

The second dataset[5] is consists of 101 data samples of animal species, each with 17

[4]http://neuroph.sourceforge.net/tutorials/SportsPrediction/download/Data%20sets.rar
[5]http://archive.ics.uci.edu/ml/datasets/Zoo

boolean attributes and one categorical, each determining a trait, some biometrics like if it has hair, feathers or tooth, or if it is a predator or domestic. (The categorical attribute is the number of legs!) It is also normalized [6] to [0, 1]. The samples are labeled, again via a vector of 0s and a single 1, into 7 classes of species' families.

[6]if it was not because of the only non boolean attribute, there would be no need for normalization

# Chapter 2

# Experiments

## 2.1 The problem and goals of experiment

When utilizing MLP and backpropagation for classification problems, like many other machine learning schemes, there are *meta-parameters* that must be set manually by the user and the learning method can not estimate them itself. It is indeed a disadvantage of a learning method, specially when these meta-parameters have significant effect on functionality of that method for different problem instances and can not be estimated (or approximated, at least) generally. So it is a common practice to *tune-up* a learning scheme for a specific problem, that is to estimate (empirically or heuristically) a set of *optimal* values for these meta-parameters, in terms of a defined criterion. And that's what we are doing in this report.

As we have seen in previous chapter, MLPs have a lot of these meta-parameters. In fact we can divide its meta-parameters into two parts: Those which determine the network's topology, and those that arise from the learning method we use; We will discuss them here in brief.

As mentioned in the last chapter, MLP is a general function approximator. Despite this can be proven theoretically, its proof will rather be *existential*, not *constructive*. So the theory won't tell us anything about how this network should look like[1] for different problems. Thus we can always add as many hidden layers with as many hidden neurons in each and obtain good results. But, for the sake of computational resources, which could be crucial in real world big or online problems, we would prefer a smaller and simpler network. Network size probably won't bother us on usual network runs, where all calculations are simple as addition, multiplication and applying activation function, but it will have significant effects on the learning phase, when the network has to take expensive computations for each layer. Specially notice we will have to compute derivatives for activation functions of all neurons. Although using exponential functions (like sigmoid and tanh, discussed before) that give us their derivatives with less effort is helpful here, but it is a costly job, and its cost grows with the network's size. So basically, it is a trade-off as usual, between the network's functionality and the computational resources. In this report, we employed an exhaustive search over different network topologies in an increasing size manner.

We also saw three learning meta-parameters in previous chapter: step size, shown by $\eta$, momentum factor, $\alpha$ and stopping criterion, which was written as $\epsilon$. In this report, we

[1]Of course, except the terminal layers which, as mentioned, are determined by the problem.

have set a fixed value for $\epsilon$ and tried to find two other meta-parameters empirically.

We also need to give a formal definition of an *optimal functionality* of a neural network for our problem, which is an instance of classification problem. Main functionality of a neural network classifier is its capability to generalize. A common benchmark for evaluation of this trait is *accuracy* the network in testing phase. It is defined as the ratio of truly classified patterns to the whole patterns in the *testing dataset*; A portion of the data which were not involved in training of the network. Accuracy is taken from subtracting *confusion*, the bottom-right-most element in the *confusion matrix*, from 1. It must be mentioned that we repeat the experiment with the same setting, shuffling the training data and split a new test dataset each time, and calculate the average of the accuracy value, in order to cancel the possible effect of random initialization or bias to specific pattern orders.

Therefore, we are looking for a setting of these meta-parameters, which give the best accuracy for the test phase and keep the network size as small as possible.

## 2.2 Methodology

It's apparent that the search space for the meta-parameters is so vast that we can not be sure if we have found an optimal setting for them, unless we use some *meta-learning* method that does this meta-optimization problem. Since we are going to find this sub-optimal setting empirically, we have to do an exhaustive search in order to increase the chance of finding an optimal setting.

There are also some experimental recommendations for these meta-parameters, some of which are included in the text book, that can give us a clue about where to look for best settings.

What is done in this project is as follows:
A function $f$ takes $\eta$, $\alpha$, $r$, $M$ and an array of integers $n_1, n_2, ..., n_k$ as parameters. $f$ creates an MLP $N$ with $k$ layers, with each layer $i$ having $n_i$ neurons. $N$ is then trained by 70% of the dataset, by backpropagation of error algorithm with momentum term, using $\eta$ and $\alpha$ as step size and momentum factor, respectively. The stopping criterion for the learning phase is one of these three condition: Error function value falls below constant $\epsilon = 0.001$, total iterations of backpropagation exceeds input parameter $M$, or no change was made in the error function in two consecutive iterations (it is below minimum floating point value). The other 30% of the data set is then fed to the network and the output of network is taken as the element which has the maximum value in the network output vector. Accuracy of one iteration is then calculated. This whole process is repeated $r$ times and average accuracy is calculated at the end. Then $f$ would give average accuracy, number of iterations of learning and it's input parameters and some other useful stuff as output (in fact output from each run of $f$ is printed in a file).

The main program has loops that goes through different input parameters of $f$, in order to search the meta-parameters space. After some hundreds of runs of the main program (literally) with different settings, a huge pile of output data is gathered as raw files. Then the best results were filtered out and are shown here in this report.

## 2.3 Remarks

Some issues were encountered during the experiments and were not discussed throughly at the class, so seem worth to be mentioned here:

- We concluded (I didn't have any other option anyways!) that in the first benchmark, there is a little (if none at all) relation between available parameters and classes, so the accuracy in the test result did not exceed much further than 60%. It seems there is still many unknown parameters in a game of football that makes watching it exciting. But its results are still shown in this report.

- To make this search for meta-parameters as exhaustive as possible, we have not imitated some proposed *greedy* methods that searches blindly and always find a good solution out of luck. Nor the proposed values (like $\eta = 0.3$ or $\alpha = 0.7$) satisfied a good result at first. Conversely, I tried to relax some criteria at first to let the program have a wider search in less time; Like giving more space to loop bounds and increasing loop steps. Specially for the first benchmark that had no success in reaching $\epsilon$ criterion in first couple of million iterations of backprop (which really wasted a good portion of my time). I chose a fixed lower value for $M$(50000 at first), giving the search a boost that let it go through the search space rapidly.

- After finding some *interesting spots* on the meta-parameter search space (were more good results were obtained), I made the search more precise at those spot to see if I can find an optimal setting around there. So the whole process was not so automatic and many user interactions were made to direct the search.

- Loops that iterated over the topology parameters $(n_1, n_2, ..., n_k)$ where the most exterior. Thus they were not at priority. But I knew that if there is not enough neurons for the problem, there won't be any optimal step size or momentum factor to find.[2] So from time to time, when I sensed[3] that maybe there are not enough neurons or layers, I would stop the program to change the outer loops initial values.

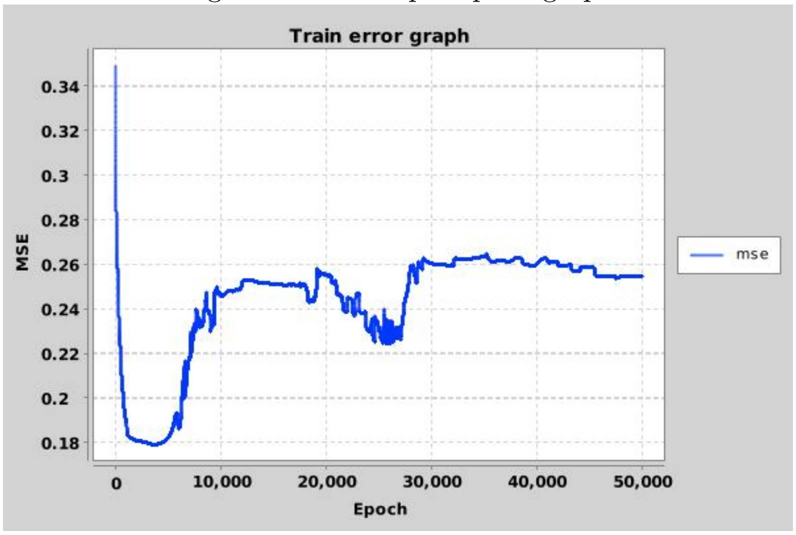
- As mentioned for the first benchmark, error value would never reach the stopping criterion and seemed to stay fixed at some point, so it would continue to reach learning iteration limit. And conversely, the few that would reach the $\epsilon$ by luck, would do that in first few iterations. I checked compared some of their error graphs, and something that caught my attention was that in many of them, the error had gone down with a fast slope, then if it hadn't reached the $\epsilon$ in first few thousand iterations, it would go up again and fix at some point, ad infinitum it seemed. It wouldn't even keep it's own minimum error it founded. You can see such graph, which was very common, in figure 2.1. I didn't have any analysis for the phenomenon back then, but I decided to keep the networks minimum error state of weights myself and later compare it's accuracy with the last state of the network's weight, after learning was done. So I changed the code again, made it so it would keep a snapshot of network weight's for the minimum value of the learning error. I hoped it could help to improve accuracy, but It didn't. The "Elite Accuracy"[4] was even lesser that

[2]It was not a necessity, but any other parameters could have this problem if they were iterated in the exterior loops. Though it seems more rational do conversely now, because at least I had some suggestions about other parameters, but got no idea about the optimal topology.

[3]From the results, of course.

[4]Since everything was already becoming like a genetic algorithm *meta-learning*, I chose this name because of it's relevance and also as a pun.

Figure 2.1: Error per epoch graph



the last accuracy many times. Couple of days-wasted later, I found why.

- I found a little delicate point we never discussed at the class. The Java library I used for backpropagation (Neuroph[5]) was training the patterns to the network in *online* mode by default, what I never thought about before. In online mode backpropagation, it would apply the $\Delta w$ that it had computed for each pattern immediately; An approach for situations that we don't have a integrated dataset, but individual data patterns which may arrive at any time, which was not our situation, absolutely. Unlike *batch* mode, which computes $\Delta w$s for all patterns in the dataset, and then apply the summation of all of them at once, which counts as an *epoch*. So I changed it to work in batch mode, and suddenly all the graphs became like figure 2.2. Now most of the runs would reach the minimum error stopping criterion in few epochs, and it was a great boost to the search speed.[6] Now I could be sure that the network is really learning the dataset. But the main problem was still there; The accuracy would not increase.

- I don't have any analysis for the results of the second bench mark. Nor it required much effort to give good results. I just ran it and it was all good like that. Even an SLP would be enough for it. It shows there are problems that are so obvious that we may decide not to bother a neural network for them, lots of simpler method probably suffice. It also shows how you can be lucky and take all the good results (and scores).

- The source code for this project, including this report, can be found at `https://github.com/BelegCuthalion/mlp-rbf-tuner`

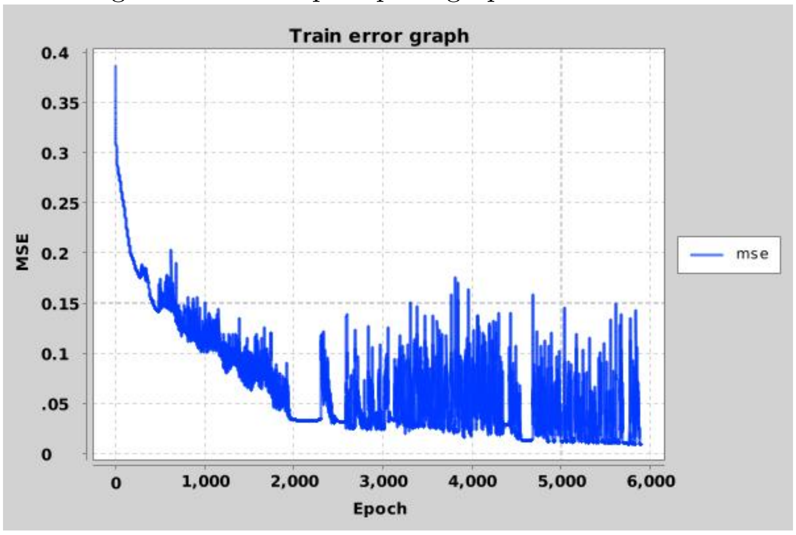
## 2.4 Results

Best results for tuning up an MLP for two mentioned benchmarks are provided in this sections. Best results for the first benchmark gives accuracy of 69% with a single layer

[5]http://neuroph.sourceforge.net/ , a Java open source neural network framework, so much better than what the author advertises in the textbook and hides it's source code

[6]It was already couple of weeks after the presentation due time though.

Figure 2.2: Error per epoch graph in batch mode



of 65 neurons. It's interesting that adding more layers does not help accuracy to go any further. And for the second benchmark we have wonderful accuracy of 98%, leads us to a conclusion that maybe it didn't need neural network classifier at all. Many cheaper method probably could take care of it.
Notice that tables are sorted by the "Elite Accuracy" column.

Columns of the tables are:

**Topology** Describes the topology of the network, from left to right, dash-separated numbers are neurons at the input, hidden and output layers.

$\eta$ Step size used in backpropagation algorithm.

$\alpha$ Momentum factor used in backpropagation algorithm.

**Last Error** The last error value when the learning algorithm stopped.

**Last Accuracy** Accuracy of network in classification of test dataset, with the last synaptic weights trained.

**Elite Error** The minimum error value during the training phase.

**Elite Accuracy** Accuracy of network in classification of test dataset, with the snapshot of the synaptic weights when network was in Elite Error state.

**A/n** A proposed parameter for evaluating the network's functionality; Accuracy divided by total number of hidden neurons, which is in fact more meaningful for RBF learning benchmarks.

Table 2.1: Benchmark 1: One layer topologies

| Topology | $\eta$ | $\alpha$ | Last Error | Last Accuracy | Elite Error | Elite Accuracy | A/n |
|---|---|---|---|---|---|---|---|
| -8-65-3 | 0.40 | 0.70 | 0.01 | 0.69 | 0.01 | 0.69 | 1.09 |
| -8-77-3 | 0.20 | 0.70 | 0.01 | 0.68 | 0.01 | 0.68 | 0.88 |
| -8-15-3 | 0.30 | 0.60 | 0.01 | 0.65 | 0.01 | 0.65 | 4.30 |
| -8-19-3 | 0.30 | 0.80 | 0.27 | 0.52 | 0.06 | 0.63 | 2.72 |
| -8-22-3 | 0.30 | 0.80 | 0.01 | 0.54 | 0.01 | 0.63 | 2.93 |
| -8-24-3 | 0.30 | 0.60 | 0.01 | 0.56 | 0.01 | 0.63 | 2.69 |
| -8-31-3 | 0.20 | 0.70 | 0.01 | 0.43 | 0.01 | 0.62 | 2.08 |
| -8-33-3 | 0.30 | 0.80 | 0.01 | 0.62 | 0.01 | 0.62 | 1.96 |
| -8-34-3 | 0.40 | 0.60 | 0.01 | 0.61 | 0.01 | 0.61 | 1.90 |
| -8-41-3 | 0.20 | 0.70 | 0.01 | 0.61 | 0.01 | 0.61 | 1.57 |
| -8-49-3 | 0.30 | 0.70 | 0.01 | 0.61 | 0.01 | 0.60 | 1.32 |
| -8-50-3 | 0.30 | 0.70 | 0.01 | 0.60 | 0.01 | 0.60 | 1.29 |
| -8-52-3 | 0.20 | 0.70 | 0.01 | 0.60 | 0.01 | 0.60 | 1.24 |
| -8-54-3 | 0.30 | 0.60 | 0.01 | 0.60 | 0.01 | 0.60 | 1.19 |
| -8-55-3 | 0.40 | 0.60 | 0.01 | 0.60 | 0.01 | 0.60 | 1.17 |
| -8-58-3 | 0.40 | 0.80 | 0.01 | 0.60 | 0.01 | 0.60 | 1.11 |

Table 2.2: Benchmark 1: Two layer topologies

| Topology | $\eta$ | $\alpha$ | Last Error | Last Accuracy | Elite Error | Elite Accuracy | A/n |
|---|---|---|---|---|---|---|---|
| -8-2-18-3 | 0.10 | 0.70 | 0.33 | 0.38 | 0.16 | 0.60 | 1.90 |
| -8-2-45-3 | 0.10 | 0.70 | 0.31 | 0.33 | 0.21 | 0.60 | 0.71 |
| -8-2-4-3 | 0.10 | 0.70 | 0.29 | 0.52 | 0.18 | 0.57 | 8.73 |
| -8-2-6-3 | 0.10 | 0.70 | 0.31 | 0.33 | 0.19 | 0.57 | 4.17 |
| -8-2-31-3 | 0.10 | 0.70 | 0.33 | 0.40 | 0.16 | 0.57 | 1.23 |
| -8-2-45-3 | 0.10 | 0.70 | 0.32 | 0.50 | 0.25 | 0.57 | 1.06 |
| -8-2-49-3 | 0.10 | 0.70 | 0.33 | 0.43 | 0.17 | 0.57 | 0.84 |
| -8-2-21-3 | 0.10 | 0.70 | 0.32 | 0.36 | 0.20 | 0.55 | 1.55 |
| -8-2-31-3 | 0.10 | 0.70 | 0.31 | 0.40 | 0.21 | 0.55 | 1.23 |
| -8-2-43-3 | 0.10 | 0.70 | 0.29 | 0.36 | 0.18 | 0.55 | 0.79 |
| -8-2-44-3 | 0.10 | 0.70 | 0.31 | 0.40 | 0.21 | 0.55 | 0.88 |
| -8-2-47-3 | 0.10 | 0.70 | 0.30 | 0.45 | 0.19 | 0.55 | 0.92 |
| -8-2-49-3 | 0.10 | 0.70 | 0.31 | 0.36 | 0.22 | 0.55 | 0.70 |
| -8-2-15-3 | 0.10 | 0.70 | 0.30 | 0.45 | 0.21 | 0.52 | 2.66 |
| -8-2-23-3 | 0.10 | 0.70 | 0.29 | 0.55 | 0.25 | 0.52 | 2.19 |
| -8-2-25-3 | 0.10 | 0.70 | 0.28 | 0.43 | 0.20 | 0.52 | 1.59 |
| -8-2-30-3 | 0.10 | 0.70 | 0.30 | 0.36 | 0.16 | 0.52 | 1.12 |
| -8-2-33-3 | 0.10 | 0.70 | 0.32 | 0.40 | 0.18 | 0.52 | 1.16 |
| -8-2-41-3 | 0.10 | 0.70 | 0.30 | 0.33 | 0.12 | 0.52 | 0.78 |
| -8-2-45-3 | 0.10 | 0.70 | 0.30 | 0.45 | 0.16 | 0.52 | 0.96 |
| -8-8-15-3 | 0.10 | 0.70 | 0.23 | 0.57 | 0.01 | 0.52 | 2.48 |

Table 2.3: Benchmark 1: Three layer topologies

| Topology | $\eta$ | $\alpha$ | Last Error | Last Accuracy | Elite Error | Elite Accuracy | A/n |
|---|---|---|---|---|---|---|---|
| -8-10-13-15-3 | 0.30 | 0.70 | 0.33 | 0.52 | 0.17 | 0.65 | 1.36 |
| -8-10-14-16-3 | 0.30 | 0.70 | 0.32 | 0.52 | 0.22 | 0.65 | 1.29 |
| -8-10-14-17-3 | 0.30 | 0.70 | 0.32 | 0.48 | 0.21 | 0.65 | 1.18 |
| -8-10-16-12-3 | 0.30 | 0.70 | 0.34 | 0.35 | 0.21 | 0.65 | 0.93 |
| -8-11-15-14-3 | 0.30 | 0.70 | 0.30 | 0.58 | 0.18 | 0.65 | 1.45 |
| -8-10-12-13-3 | 0.30 | 0.70 | 0.34 | 0.35 | 0.20 | 0.61 | 1.01 |
| -8-10-15-14-3 | 0.30 | 0.70 | 0.34 | 0.39 | 0.15 | 0.61 | 0.99 |
| -8-10-18-16-3 | 0.30 | 0.70 | 0.35 | 0.35 | 0.14 | 0.61 | 0.81 |
| -8-11-10-10-3 | 0.30 | 0.70 | 0.34 | 0.45 | 0.20 | 0.61 | 1.46 |
| -8-11-10-12-3 | 0.30 | 0.70 | 0.30 | 0.35 | 0.20 | 0.61 | 1.08 |
| -8-11-13-15-3 | 0.30 | 0.70 | 0.34 | 0.48 | 0.14 | 0.61 | 1.24 |
| -8-11-14-16-3 | 0.30 | 0.70 | 0.32 | 0.39 | 0.16 | 0.61 | 0.94 |
| -8-11-15-13-3 | 0.30 | 0.70 | 0.29 | 0.39 | 0.18 | 0.61 | 0.99 |
| -8-11-15-15-3 | 0.30 | 0.70 | 0.34 | 0.45 | 0.19 | 0.61 | 1.10 |

Table 2.4: Benchmark 2

| Topology | $\eta$ | $\alpha$ | Last Error | Last Accuracy | Elite Error | Elite Accuracy | A/n |
|---|---|---|---|---|---|---|---|
| -16-15-7 | 0.20 | 0.70 | 0.01 | 0.98 | 0.01 | 0.98 | 6.57 |
| -16-16-7 | 0.10 | 0.70 | 0.01 | 0.97 | 0.01 | 0.97 | 6.07 |
| -16-17-7 | 0.20 | 0.70 | 9.95E-4 | 0.96 | 9.95E-4 | 0.96 | 5.63 |
| -16-18-7 | 0.20 | 0.70 | 9.89E-4 | 0.96 | 9.89E-4 | 0.96 | 5.32 |
| -16-13-7 | 0.10 | 0.90 | 9.93E-4 | 0.94 | 9.93E-4 | 0.94 | 7.25 |
| -16-7-7 | 0.20 | 0.70 | 0.01 | 0.94 | 0.01 | 0.94 | 13.47 |
| -16-9-7 | 0.20 | 0.70 | 0.02 | 0.94 | 0.02 | 0.94 | 10.48 |
| -16-14-7 | 0.10 | 0.70 | 9.90E-4 | 0.93 | 9.90E-4 | 0.93 | 6.63 |
| -16-17-7 | 0.10 | 0.70 | 9.87E-4 | 0.93 | 9.87E-4 | 0.93 | 5.46 |
| -16-18-7 | 0.10 | 0.70 | 9.93E-4 | 0.93 | 9.93E-4 | 0.93 | 5.16 |
| -16-19-7 | 0.10 | 0.70 | 0.01 | 0.93 | 0.01 | 0.93 | 4.89 |
| -16-9-7 | 0.10 | 0.80 | 9.94E-4 | 0.93 | 9.94E-4 | 0.93 | 10.32 |
| -16-10-7 | 0.10 | 0.80 | 9.92E-4 | 0.93 | 9.92E-4 | 0.93 | 9.29 |
| -16-8-7 | 0.10 | 0.90 | 0.01 | 0.93 | 0.01 | 0.93 | 11.61 |
| -16-8-7 | 0.20 | 0.70 | 0.01 | 0.93 | 0.01 | 0.93 | 11.61 |
| -16-9-7 | 0.20 | 0.80 | 9.99E-4 | 0.93 | 9.99E-4 | 0.93 | 10.32 |
| -16-13-7 | 0.20 | 0.80 | 9.86E-4 | 0.93 | 9.86E-4 | 0.93 | 7.14 |
| -16-15-7 | 0.20 | 0.80 | 9.86E-4 | 0.93 | 9.86E-4 | 0.93 | 6.19 |
| -16-13-7 | 0.10 | 0.70 | 9.93E-4 | 0.91 | 9.93E-4 | 0.91 | 7.03 |
| -16-6-7 | 0.10 | 0.80 | 0.01 | 0.91 | 0.01 | 0.91 | 15.24 |
| -16-12-7 | 0.10 | 0.90 | 0.01 | 0.91 | 0.01 | 0.91 | 7.62 |
| -16-19-7 | 0.20 | 0.70 | 9.93E-4 | 0.91 | 9.93E-4 | 0.91 | 4.81 |
| -16-11-7 | 0.20 | 0.80 | 0.01 | 0.91 | 0.01 | 0.91 | 8.31 |